Trabajo Práctico integrador - Programación con objetos 2 - UNQ

A la caza de las vinchucas

Por: Bogado Ximena - Cáceres Gonzalo - Mendez Nahuel

Introducción

Informe detallado sobre el funcionamiento de la aplicación web solicitada en la consigna de “A la caza de vinchucas” del trabajo práctico integrador de programación con objetos 2.

Equipo:

“Las tres vinchuecas”

Integrantes

- Bogado Ximena
- Caceres Gonzalo
- Mendez Nahuel

Universidad Nacional De Quilmes

Programación con Objetos 2

Índice

Diseño UML

Construcción

Para realizar el diagrama de clases, hicimos reuniones de equipo frecuentes, tratando de aplicar todo lo que vimos en las clases. En cada reunión leíamos el enunciado, tratábamos de abstraer lo que nos estaban pidiendo y discutíamos entre los tres, las posibles soluciones para satisfacer las consignas. Nos enfocamos principalmente en el diseño y dejamos la parte de codificación para después de haber terminado el diagrama.

Durante esta etapa, se tomó gran parte de la toma de decisiones, tanto de patrones de diseño a aplicar, como delegaciones y/o protocolos.

Herramientas

Para dibujar el diagrama, utilizamos la herramienta web *draw.io* recomendada por los profesores, y como fuente de consulta contamos con la referencia bibliográfica facilitada también por el equipo docente.

Clases que participan

A continuación describiremos brevemente las clases más importantes del modelo y sus roles. Hay más y se pueden observar en el archivo que contiene el diagrama de clases UML y el código implementado en java.

Muestra

La clase Muestra se encarga principalmente de la verificación y obtención de resultados a partir de las opiniones que los usuarios realizan sobre la misma con el motivo de detectar la presencia de una vinchuca en la zona en la que fue tomada la muestra. Para ello cuenta con:

- estados que permiten la transición de una muestra votada a verificada y restringir las operaciones de los usuarios en la muestra según lo amerite.
- datos recolectados a partir de la creación y operaciones que se realizan en la muestra como enviar muestras y opinar sobre muestra.

Usuario

La clase Usuario es la encargada de realizar operaciones sobre la muestra con el motivo de proveer información para la detección de la vinchuca. Las operaciones más importantes de esta clase son el envío de muestras y opiniones sobre muestras. Para ello cuenta con:

- Dos tipos de usuarios, especialista y novato. El usuario especialista siempre cuenta con un nivel de experto y el usuario novato puede cambiar de nivel dependiendo de la cantidad de envíos y revisiones.
- Dos estados de usuario, usuario básico y usuario experto. Los usuarios básicos opinan sobre la muestra sin poder validarla y los usuarios expertos opinan sobre la muestra para poder validarla.

Aplicación web

La clase AplicacionWeb es la encargada de mantener un registro de los usuarios que participan, las muestras enviadas y las zonas de cobertura, también se encarga de proveer estos datos en una colaboración con otra clase para realizar determinadas operaciones.

Ubicación

Ubicación es una clase con la que colaboran varias clases del modelo, tales como Muestra, ZonaDeCobertura y Organización con el fin de medir distancias entre sí y hacer búsquedas por cercanía.

Zona de cobertura

La clase ZonaDeCobertura, se encarga de llevar el registro de las muestras que se registran en un radio de cobertura determinado y de dar aviso a las clases interesadas cuando las mismas se registran o verifican. Para ello dichas clases interesadas deben registrarse a una zona de cobertura.

Organización

Cumple la función de *observar* a ZonaDeCobertura. Esta clase, dependiendo de lo que registre Zona de cobertura (ya sea una nueva muestra o la verificación de una muestra existente) invocará a una de sus funcionalidades externas para enviarles el mensaje 'nuevoEvento' que corresponde a la interface que implementan las funcionalidades externas.

Filtros

Para realizar las búsquedas en la aplicación web creamos clases de filtros que pedían el enunciado. La AplicaciónWeb será la clase encargada de invocarlos y pedirles que filtren una lista de muestra de acuerdo a la condición que se les otorgue. Tienen subclases y un diseño pensado para que esté abierto a la extensión y cerrado a la modificación.

Decisiones de diseño y justificaciones

Para empezar, tenemos una clase de tipo Enumerador -*Descripción*- que representa a la descripción que tiene una opinión sobre una muestra. Tomamos la decisión de hacerla de esta forma, porque, por un lado, queríamos seguir trabajando con “objetos” -era más enriquecedora la idea de trabajar con instancias de un Enumerador que con strings- ; y por otro lado, queríamos preservar valores fijos que pueda tomar una opinión. De igual manera y con el mismo fin, usamos una clase de tipo Enumerador para los tipos de Organización. Quisimos hacer un refactor para Opinión y Descripción para que sean una sola clase de tipo Enumerador, pero tuvimos la complicación de que no podíamos mockear una clase Enumerador y lo dejamos como estaba.

Con respecto a los filtros, decidimos diseñar el comportamiento de los filtros por fecha con una *estrategia* intercambiable. Esta solución fue la que más nos convenció ya que permitía extender el comportamiento del programa sin tener que modificar la implementación de alguna clase.

Fixtures de tests

Las siguientes clases Fixture fueron creadas en el código con el único motivo de testear el comportamiento de algunas clases frente a distintos escenarios, por lo tanto no están incluidas en el diagrama de clases ni en la carpeta source del código.

Estas clases fixture son:

- FixtureUsuarioNovatoTest
- FixtureFiltrosPorFechaConEstrategias
- FixtureListasDeMuestrasParaFiltros
- FixtureUsuarioNovatoParaBajarDeCategoriaTest

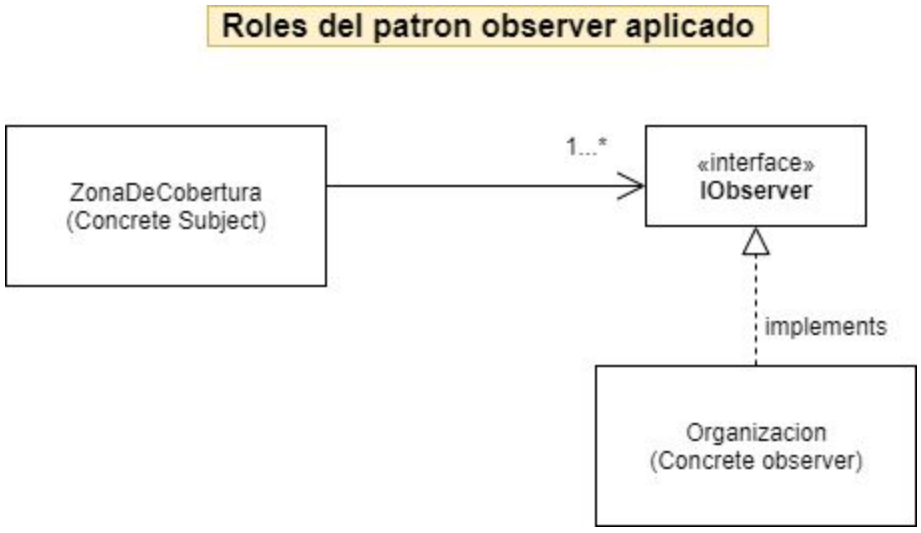
Patrones de diseño aplicados y las clases que cumplen los roles correspondientes en cada caso

A medida que fuimos avanzando en el diseño, aplicamos distintos patrones para solucionar problemas comunes. Los patrones que utilizamos en nuestra solución fueron:

- Observer Pattern
- Strategy Pattern
- Composite Pattern
- State Pattern

Observer Pattern

Decidimos usar el patrón Observer en el aviso a las organizaciones porque la Organización depende de los cambios de una ZonaDeCobertura para poder ejecutar la funcionalidad de 'nuevoEvento' que provee la interface FuncionalidadExterna. Para evitar que la Organización este constantemente preguntando si la ZonaDeCobertura "cambió", la ZonaDeCobertura le da un aviso a las clases interesadas con los cambios referentes al registro o verificación de una muestra en esa zona.



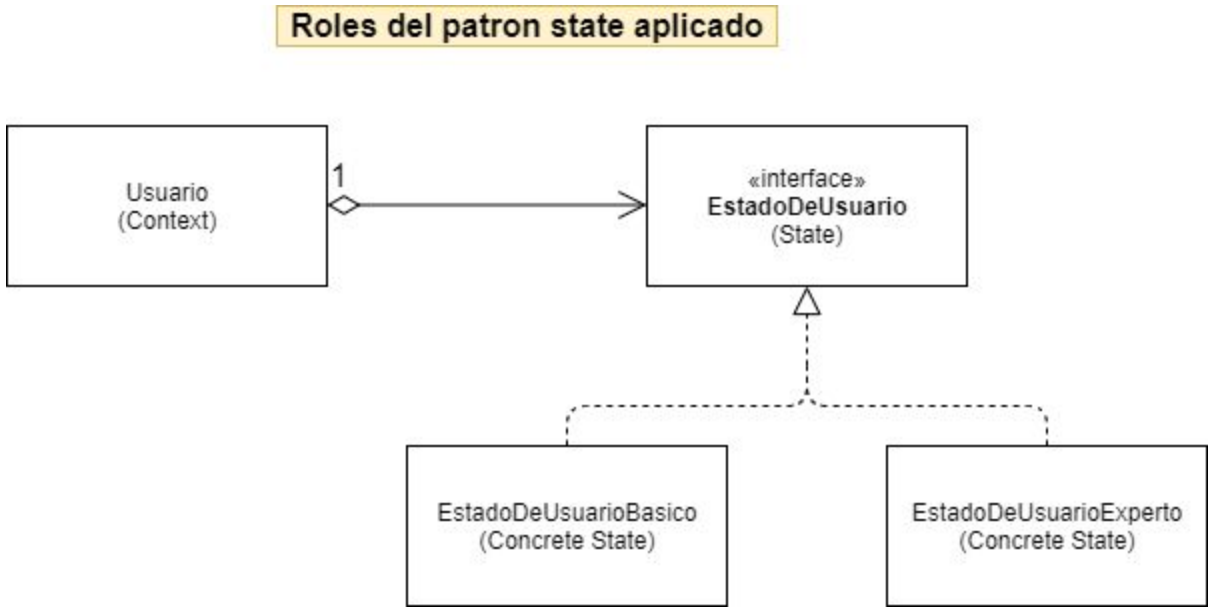
Concrete observer -> Organización

Observer -> IObserver

Concrete Subject -> ZonaDeCobertura

State Pattern en Usuario

En Usuario decidimos usar el patrón State porque el comportamiento de opinar sobre una muestra varía de acuerdo al estado actual del participante.



Context -> Usuario

State -> EstadoDeUsuario

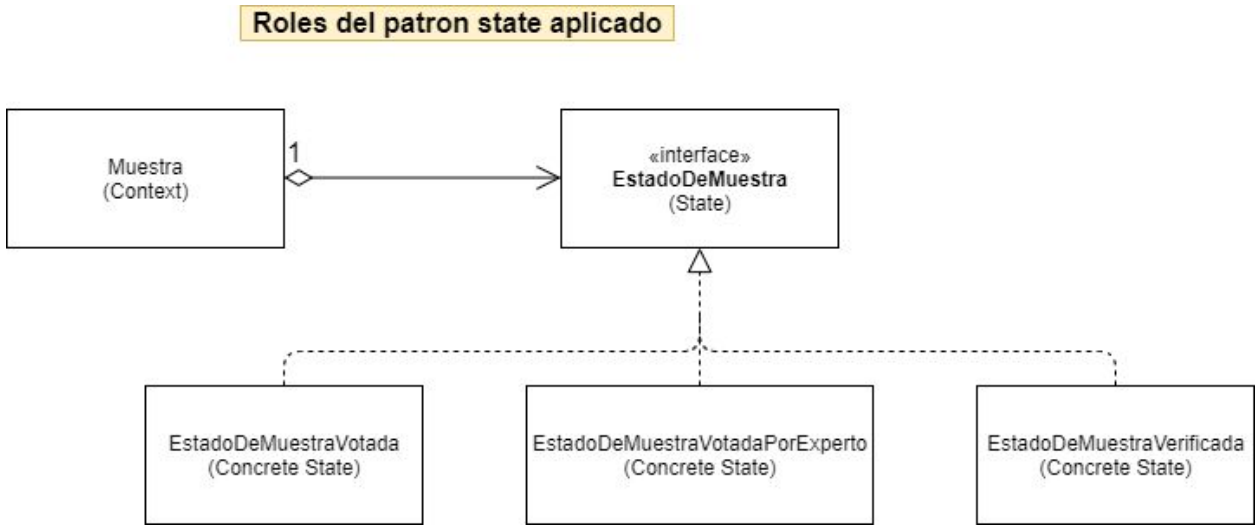
Concrete State A -> EstadoDeUsuarioBasico

Concrete State B -> EstadoDeUsuarioExperto

State Pattern en Muestra

Al igual que en la clase Usuario, la clase Muestra depende de su estado para realizar diferentes operaciones como agregar opinión y verificar muestra.

De esta manera se logra que los distintos comportamientos, al agregarse una opinión o verificarse una muestra, sean encapsulados en estados y se evita el uso y anidación de condicionales para seleccionar el comportamiento apropiado dependiendo del estado actual del objeto Muestra.



Context -> Muestra

State -> EstadoDeMuestra

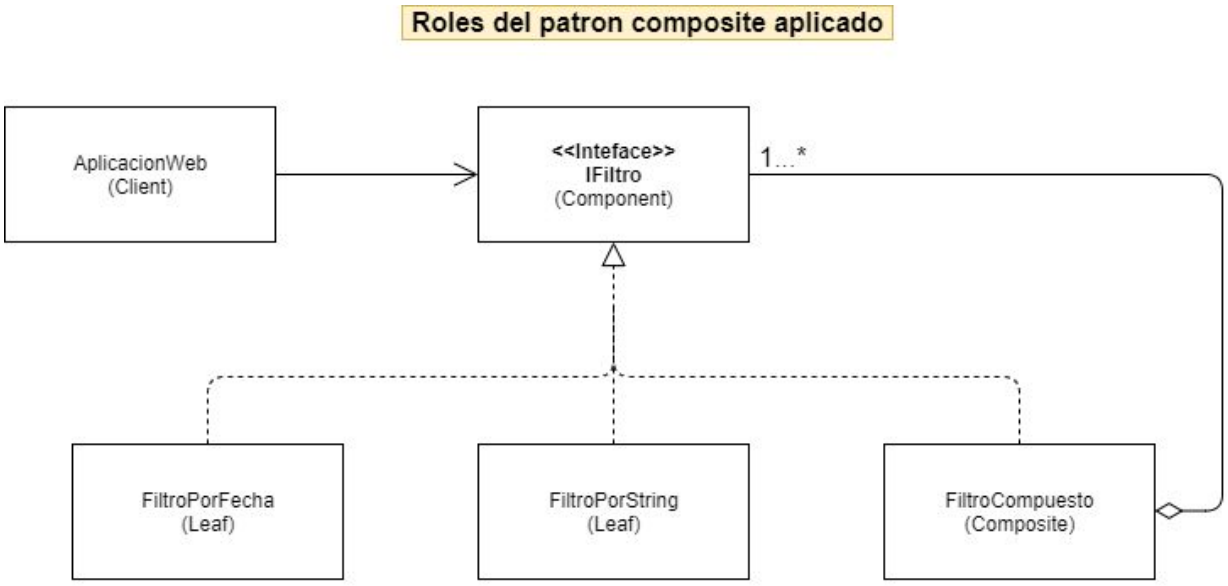
Concrete State A -> EstadoDeMuestraVotada

Concrete State B -> EstadoDeMuestraVotadaPorExperto

Concrete State C -> EstadoDeMuestraVerificada

Composite Pattern

Se eligió el patrón Composite con el motivo de que la aplicación web pudiera utilizar cada composición de filtro de manera uniforme, pudiendo así ignorar la diferencia entre composiciones de objetos e individuos objetos.



Cliente -> AplicaciónWeb

Component -> IFiltro

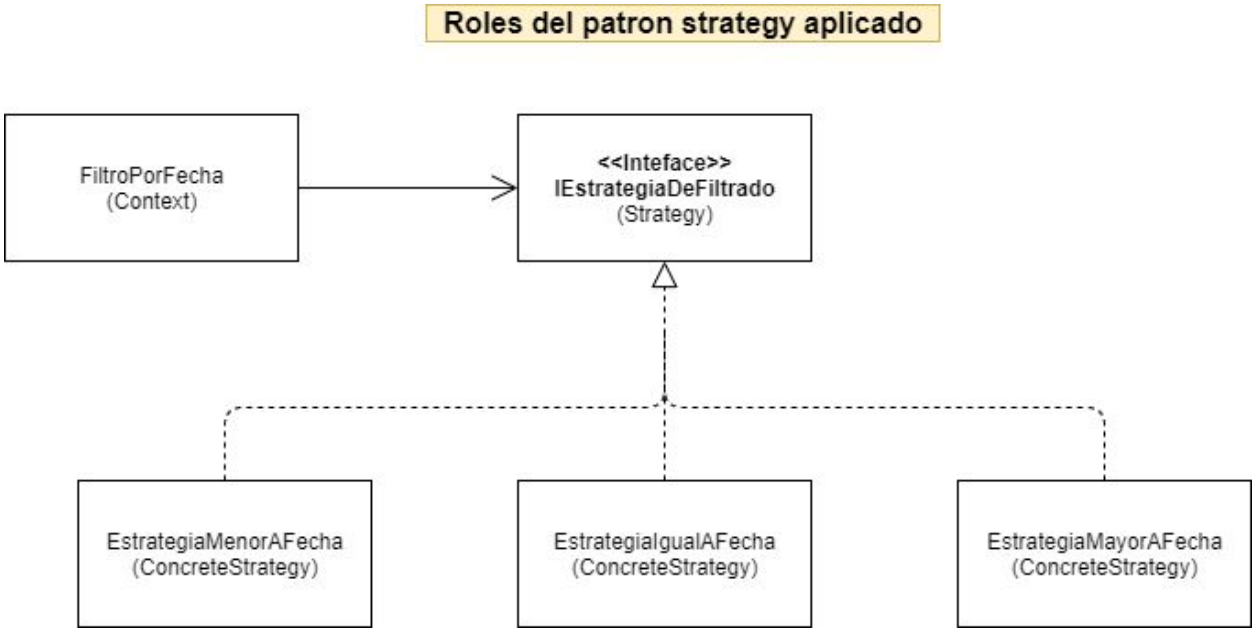
Leaf A -> FiltroPorFecha

Leaf B -> FiltroPorString

Composite -> FiltroCompuesto

Strategy Pattern

Decidimos utilizar el patrón Strategy para poder configurar un filtro simple por fecha con diferentes tipos de desigualdades (=,> y <). Esto nos da la posibilidad de intercambiar de forma dinámica el comportamiento en filtros por fecha de creación de muestra y en filtros por fecha de última votación.



Context -> FiltroPorFecha

Strategy -> IEstrategiaDeFiltrado

Concrete Strategy A -> EstrategiaMenorAFecha

Concrete Strategy B -> EstrategiaIgualAFecha

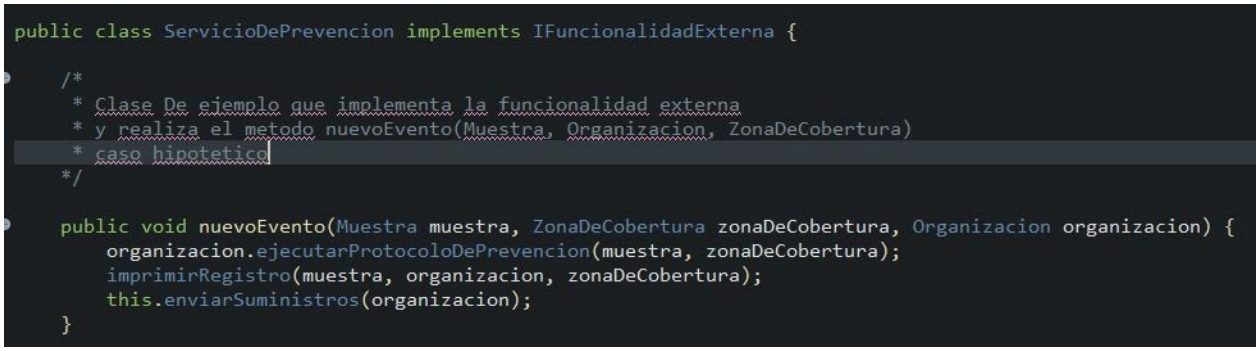
Concrete Strategy C -> EstrategiaMayorAFecha

Funcionalidad externa

Implementación y uso de la funcionalidad externa:

Una clase que implemente la funcionalidad externa, al recibir el mensaje nuevoEvento, utilizaría los datos recibidos como parámetro para realizar una acción determinada.

Por ejemplo, quien implemente la funcionalidad externa (que puede ser cualquiera que utilice la información del mensaje nuevoEvento), le avisaría a los empleados de una organización que en determinada zona se ha detectado la aparición de una vinchuca de una especie y les enviaría el protocolo correspondiente a ejecutar.



```
public class ServicioDePrevencion implements IFuncionalidadExterna {

    /*
     * Clase De ejemplo que implementa la funcionalidad externa
     * y realiza el metodo nuevoEvento(Muestra, Organizacion, ZonaDeCobertura)
     * caso hipotetico
    */

    public void nuevoEvento(Muestra muestra, ZonaDeCobertura zonaDeCobertura, Organizacion organizacion) {
        organizacion.ejecutarProtocoloDePrevencion(muestra, zonaDeCobertura);
        imprimirRegistro(muestra, organizacion, zonaDeCobertura);
        this.enviarSuministros(organizacion);
    }
```

Conclusiones

El trabajo práctico supuso todo un desafío para los tres integrantes. Partiendo por la complejidad del enunciado, y los problemas que planteaba, llevó a que le dediquemos muchisimo tiempo al diseño. El trabajo en equipo y sincronizado usando Git fue muy enriquecedor. Todo el código que implementamos esta desarrollado con TDD, tratando de cumplir con los principios SOLID en su totalidad. Gracias al uso de herramientas como Mockito logramos hacer Tests de clases que tenían dependencias con otras que a su vez, o no estaban implementadas, o se encargaba de implementar un compañero. Esto nos dio el beneficio de tener clases desacopladas y de llevar un desarrollo ágil y ordenado. Estamos muy contentos y satisfechos con la experiencia adquirida en este proyecto, intentamos de aplicar las cosas que vimos en clase, los conceptos que tenemos en el material bibliográfico facilitado por los docentes y las correcciones de los docentes que nos fueron guiando en el desarrollo.